This document explains how to build Widget and DOM structures from XML markup using UiBinder, introduced with GWT 2.0. It does not cover binder's localization features—read about them in Internationalization - UiBinder.

Overview

At heart, a GWT application is a web page. And when you're laying out a web page, writing HTML and CSS is the most natural way to get the job done. The UiBinder framework allows you to do exactly that: build your apps as HTML pages with GWT widgets sprinkled throughout them.

Besides being a more natural and concise way to build your UI than doing it through code, UiBinder can also

Terms

HTML into `innerHTML` attributes than by a bunch of API calls. UiBinder naturally takes advantage of this, and the result is that the most pleasant way to build your app is also the best way to build it.

UiBinder...

- helps productivity and maintainability — it's easy to create UI from scratch or copy/paste across templates;
- makes it easier to collaborate with UI designers who are more comfortable with XML, HTML and CSS than Java source code;
- provides a gradual transition during development from HTML mocks to real, interactive UI;
- encourages a clean separation of the aesthetics of your UI (a declarative XML template) from its programmatic behavior (a Java class);
- performs thorough compile-time checking of cross-references from Java source to XML and vice-versa;
- offers direct support for internationalization that works well with GWT's i18n facility; and
- encourages more efficient use of browser resources by making it convenient to use lightweight HTML elements rather than heavier-weight widgets and panels.

But as you learn what UiBinder is, you should also understand what it is not. It is not a renderer, or at any rate that is not its focus. There are no loops, no conditionals, no if statements in its markup, and only a very limited expression language. UiBinder allows you to lay out your user interface. It's still up to the widgets or other controllers themselves to convert rows of data into rows of HTML.

The rest of this page explains how to use UiBinder through a series of typical use cases. You'll see how to lay out a UI, how to style it, and how to attach event handlers to it. Internationalization - UiBinder explains how to internationalize it.

Quick start: If instead you want to jump right in to the code, take a peek at this patch. It includes the work to change the venerable Mail sample to use UiBinder. Look for pairs of files like Mail.java and Mail.ui.xml.

Hello World

Here's a very simple example of a UiBinder template that contains no widgets, only HTML. This may seem an odd choice for a Hello World sample, because it isn't a terribly typical way to manage your UI in GWT. But it shows us the bare nuts and bolts, and reminds us that you aren't forced to pay the widget tax just to have templates.

```
<!-- HelloWorld.ui.xml -->

<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'>
  <div>
    Hello, <span ui:field='nameSpan'/>.
  </div>
</ui:UiBinder>
```

Now suppose you need to programatically read and write the text in the span (the one with the `ui:field='nameSpan'` attribute) above. You'd probably like to write actual Java code to do things like that, so UiBinder templates have an associated owner class that allows programmatic access to the UI constructs

declared in the template. An owner class for the above template might look like this:

```
public class HelloWorld {
  interface MyUiBinder extends UiBinder<DivElement, HelloWorld> {}
  private static MyUiBinder uiBinder = GWT.create(MyUiBinder.class);

  @UiField SpanElement nameSpan;

  private DivElement root;

  public HelloWorld() {
    root = uiBinder.createAndBindUi(this);
  }

  public Element getElement() {
    return root;
  }

  public void setName(String name) { nameSpan.setInnerText(name); }
}
```

You then instantiate and use the owner class as you would any other chunk of UI code. We'll see examples later that demonstrate how to use widgets with UiBinder, but this example uses direct DOM manipulation:

```
HelloWorld helloWorld = new HelloWorld();
// Don't forget, this is DOM only; will not work with GWT widgets
Document.get().getBody().appendChild(helloWorld.getElement());
helloWorld.setName("World");
```

UiBinder instances are factories that generate a UI structure and glue it to an owning Java class. The `UiBinder<U, O>` interface declares two parameter types:

- `U` is the type of root element declared in the ui.xml file, returned by the `createAndBindUi` call
- `O` is the owner type whose `@UiField`s are to be filled in.

(In this example U is DivElement and 0 is HelloWorld.)

Any object declared in the ui.xml file, including any DOM elements, can be made available to the owning Java class through its field name. Here, a `<span>` element in the markup is given a `ui:field` attribute set to nameSpan. In the Java code, a field with the same name is marked with the `@UiField` annotation. When `uiBinder.createAndBindUi(this)` is run, the field is filled with the appropriate instance of SpanElement.

Our HelloWorld object is nothing special, it has no superclass. But it could just as easily extend UIObject. Or Widget. Or Composite. There are no restrictions. However, do note that the fields marked with @UiField have default visibility. If they are to be filled by a binder, they cannot be private.

Hello Widget World

Here's an example of a UiBinder template that uses widgets:

```
<!-- HelloWidgetWorld.ui.xml -->

<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
    xmlns:g='urn:import:com.google.gwt.user.client.ui'>
```

```
  <g:HTMLPanel>
    Hello, <g:ListBox ui:field='listBox' visibleItemCount='1'/>.
  </g:HTMLPanel>

</ui:UiBinder>
```

```
public class HelloWidgetWorld extends Composite {

  interface MyUiBinder extends UiBinder<Widget, HelloWidgetWorld> {}
  private static MyUiBinder uiBinder = GWT.create(MyUiBinder.class);

  @UiField ListBox listBox;

  public HelloWidgetWorld(String... names) {
    // sets listBox
    initWidget(uiBinder.createAndBindUi(this));
    for (String name : names) {
      listBox.addItem(name);
    }
  }
}

// Use:

HelloWidgetWorld helloWorld =
  new HelloWidgetWorld("able", "baker", "charlie");
```

Note that we're using widgets, and also creating a widget. The HelloWorldWidget can be added to any panel class.

In order to use a set of widgets in a ui.xml template file, you need to tie their package to an XML namespace prefix. That's what's happening in this attribute of the root <ui:uibinder> element: `xmlns:g='urn:import:com.google.gwt.user.client.ui'`. This says that every class in the `com.google.gwt.user.client.ui` package can be used as an element with prefix g and a tag name matching its Java class name, like `<g:ListBox>`.

See how the `g:ListBox` element has a `visibleItemCount='1'` attribute? That becomes a call to ListBox#setVisibleItemCount(int). Every one of the widget's methods that follow JavaBean-style conventions for setting a property can be used this way.

Pay particular attention to the use of an HTMLPanel instance. HTMLPanel excels at mingling arbitrary HTML and widgets, and UiBinder works very well with HTMLPanel. In general, any time you want to use HTML markup inside of a widget hierarchy, you'll need an instance of HTMLPanel or the HTML Widget.

Using Panels

Any panel (in theory, anything that implements the HasWidgets interface) can be used in a template file, and can have other panels inside of it.

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
    xmlns:g='urn:import:com.google.gwt.user.client.ui'>

  <g:HorizontalPanel>
    <g:Label>Keep your ducks</g:Label>
    <g:Label>in a row</g:Label>
  </g:HorizontalPanel>

</ui:UiBinder>
```

Some stock GWT widgets require special markup, which you'll find described in their javadoc. Here's how DockLayoutPanel works:

```
<g:DockLayoutPanel unit='EM'>
  <g:north size='5'>
    <g:Label>Top</g:Label>
  </g:north>
  <g:center>
    <g:Label>Body</g:Label>
  </g:center>
  <g:west size='10'>
    <g:HTML>
      <ul>
        <li>Sidebar</li>
        <li>Sidebar</li>
        <li>Sidebar</li>
      </ul>
    </g:HTML>
  </g:west>
</g:DockLayoutPanel>
```

The DockLayoutPanel's children are gathered in organizational elements like `<g:north>` and `<g:center>`. Unlike almost everything else that appears in the template, they do not represent runtime objects. You can't give them `ui:field` attributes, because there would be nothing to put in the field in your Java class. This is why their names are not capitalized, to give you a clue that they're not "real". You'll find that other special non-runtime elements follow the same convention.

Another thing to notice is that we can't put HTML directly in most panels, but only in widgets that know what to do with HTML, specifically, HTMLPanel, and widgets that implement the HasHTML interface (such as the sidebar under `<g:west>`). Future releases of GWT will probably drop this restriction, but in the meantime it's up to you to place your HTML into HTML-savvy widgets.

HTML entities

UiBinder templates are XML files, and XML doesn't understand entities like ` `. When you need such characters, you have to define them yourself. As a convenience, we provide a set of definitions that you can import by setting your DOCTYPE appropriately:

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
```

Note that the GWT compiler won't actually visit this URL to fetch the file, because a copy of it is baked into the compiler. However, your IDE may fetch it.

Simple binding of event handlers

One of UiBinder's goals is to reduce the tedium of building user interfaces in Java code, and few things in Java require more mind-numbing boilerplate than event handlers. How many times have you written something like this?

```
public class MyFoo extends Composite {
  Button button = new Button();

  public MyFoo() {
    button.addClickHandler(new ClickHandler() {
```

```
      public void onClick(ClickEvent event) {
        handleClick();
      }
    });
    initWidget(button);
  }

  void handleClick() {
    Window.alert("Hello, AJAX");
  }
}
```

In a UiBinder owner class, you can use the `@UiHandler` annotation to have all of that anonymous class nonsense written for you.

```
public class MyFoo extends Composite {
  @UiField Button button;

  public MyFoo() {
    initWidget(button);
  }

  @UiHandler("button")
  void handleClick(ClickEvent e) {
    Window.alert("Hello, AJAX");
  }
}
```

However, there is one limitation (at least for now): you can only use `@UiHandler` with events thrown by widget objects, not DOM elements. That is, `<g:Button>`, not `<button>`.

Using a widget that requires constructor args

Every widget that is declared in a template is created by a call to `GWT.create()`. In most cases this means that they must be default instantiable; that is, they must provide a zero-argument constructor. However, there are a few ways to get around that. In addition to the `@UiFactory` and `@UiField(provided = true)` mechanisms described under Shared resource instances, you can mark your own widgets with the `@UiConstructor` annotation.

Suppose you have an existing widget that needs constructor arguments:

```
public CricketScores(String... teamNames) {...}
```

You use it in a template:

```
<!-- UserDashboard.ui.xml -->

<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
    xmlns:g='urn:import:com.google.gwt.user.client.ui'
    xmlns:my='urn:import:com.my.app.widgets' >

  <g:HTMLPanel>
    <my:WeatherReport ui:field='weather'/>

    <my:Stocks ui:field='stocks'/>
    <my:CricketScores ui:field='scores' />
```

```
  </g:HTMLPanel>
</ui:UiBinder>
```

```
public class UserDashboard extends Composite {
  interface MyUiBinder extends UiBinder<Widget, UserDashboard> {}
  private static MyUiBinder uiBinder = GWT.create(MyUiBinder.class);

  public UserDashboard() {
    initWidget(uiBinder.createAndBindUi(this));
  }
}
```

An error results:

```
[ERROR] com.my.app.widgets.CricketScores has no default (zero args)
constructor. To fix this, you can define a @UiFactory method on the
UiBinder's owner, or annotate a constructor of CricketScores with
@UiConstructor.
```

So you either make the @UiFactory method...

```
public class UserDashboard extends Composite {
  interface MyUiBinder extends UiBinder<Widget, UserDashboard>;
  private static final MyUiBinder uiBinder = GWT.create(MyUiBinder.class);

  private final String[] teamNames;

  public UserDashboard(String... teamNames) {
    this.teamNames = teamNames;
    initWidget(uiBinder.createAndBindUi(this));
  }

  /** Used by MyUiBinder to instantiate CricketScores */
  @UiFactory CricketScores makeCricketScores() { // method name is insignificant
    return new CricketScores(teamNames);
  }
}
```

...annotate a constructor...

```
public @UiConstructor CricketScores(String teamNames) {
  this(teamNames.split("[, ]+"));
}
```

```
<!-- UserDashboard.ui.xml -->
<g:HTMLPanel xmlns:ui='urn:ui:com.google.gwt.uibinder'
  xmlns:g='urn:import:com.google.gwt.user.client.ui'
  xmlns:my='urn:import:com.my.app.widgets' >

  <my:WeatherReport ui:field='weather'/>
  <my:Stocks ui:field='stocks'/>
  <my:CricketScores ui:field='scores' teamNames='AUS, SAF, WA, QLD, VIC'/>

</g:HTMLPanel>
```

...or fill in a field marked with `@UiField(provided=true)`

```
public class UserDashboard extends Composite {
  interface MyUiBinder extends UiBinder<Widget, UserDashboard>;
  private static final MyUiBinder uiBinder = GWT.create(MyUiBinder.class);

  @UiField(provided=true)
  final CricketScores cricketScores; // cannot be private

  public UserDashboard(CricketScores cricketScores) {
    // DI fans take note!
    this.cricketScores = cricketScores;
    initWidget(uiBinder.createAndBindUi(this));
  }
}
```

Hello Stylish World

With the <ui:style> element, you can define the CSS for your UI right where you need it.

Note: <ui:style> elements must be direct children of the root element. The same is true of the other resource elements (<ui:image> and <ui:data>).

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'>

  <ui:style>
    .pretty { background-color: Skyblue; }
  </ui:style>

  <div class='{style.pretty}'>
    Hello, <span ui:field='nameSpan'/>.
  </div>

</ui:UiBinder>
```

A CssResource interface is generated for you, along with a ClientBundle. This means that the compiler will warn you if you misspell the class name when you try to use it (e.g. {style.prettty}). Also, your CSS class name will be obfuscated, thus protecting it from collision with like class names in other CSS blocks—no more global CSS namespace!

In fact, you can take advantage of this within a single template:

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'>
  <ui:style>
    .pretty { background-color: Skyblue; }
  </ui:style>

  <ui:style field='otherStyle'>
    .pretty { background-color: Orange; }
  </ui:style>

  <div class='{style.pretty}'>
    Hello, <span class='{otherStyle.pretty}' ui:field='nameSpan'/>.
  </div>

</ui:UiBinder>
```

Finally, you don't have to have your CSS inside your ui.xml file. Most real world projects will probably keep their CSS in a separate file. In the example given below, the src values are relative to the location of the ui.xml file.

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'>
  <ui:style src="MyUi.css" />
  <ui:style field='otherStyle' src="MyUiOtherStyle.css">

  <div class='{style.pretty}'>
    Hello, <span class='{otherStyle.pretty}' ui:field='nameSpan'/>.
  </div>
</ui:UiBinder>
```

And you can set style on a widget, not just HTML. Use the `styleName` attribute to override whatever CSS styling the widget defaults to (just like calling setStyleName() in code). Or, to add class names without clobbering the widget's baked in style settings, use the special `addStyleNames` attribute:

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
      xmlns:g='urn:import:com.google.gwt.user.client.ui'>
  <ui:style>
    .hot { color: magenta; }
    .pretty { background-color: Skyblue; }
  </ui:style>

  <g:PushButton styleName='{style.pretty}'>This button doesn't look like one</g:PushButton>
  <g:PushButton addStyleNames='{style.pretty} {style.hot}'>Push my hot button!</g:PushButton>

</ui:UiBinder>
```

Note that addStyleNames is plural.

Programmatic access to inline Styles

Your code will need access to at least some of the styles your template uses. For example, suppose your widget needs to change color when it's enabled or disabled:

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'>

  <ui:style type='com.my.app.MyFoo.MyStyle'>
    .redBox { background-color:pink; border: 1px solid red; }
    .enabled { color:black; }
    .disabled { color:gray; }
  </ui:style>

  <div class='{style.redBox} {style.enabled}'>I'm a red box widget.</div>

</ui:UiBinder>
```

```
public class MyFoo extends Widget {
  interface MyStyle extends CssResource {
    String enabled();
    String disabled();
  }

  @UiField MyStyle style;

  /* ... */

  void setEnabled(boolean enabled) {
    getElement().addClassName(enabled ? style.enabled() : style.disabled());
```

```
    getElement().removeClassName(enabled ? style.disabled() : style.enabled());
  }
}
```

The <ui:style> element has a new attribute, type='com.my.app.MyFoo.MyStyle'. That means that it needs to implement that interface (defined in the Java source for the MyFoo widget above) and provide the two CSS classes it calls for, enabled and disabled.

Now look at the @UiField MyStyle style; field in MyFoo.java. That gives the code access to the CssResource generated for the <ui:style> block. The setEnabled method uses that field to apply the enabled and disabled styles as the widget is turned on and off.

You're free to define as many other classes as you like in a style block with a specified type, but your code will have access only to those required by the interface.

Using an external resource

Sometimes your template will need to work with styles or other objects that come from outside of your template. Use the <ui:with> element to make them available.

```
<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'
    xmlns:g='urn:import:com.google.gwt.user.client.ui'>

  <ui:with field='res' type='com.my.app.widgets.logoname.Resources'/>

  <g:HTMLPanel>

    <g:Image resource='{res.logo}'/>

    <div class='{res.style.mainBlock}'>
      <div class='{res.style.userPictureSprite}'/>

      <div>
        Well hello there
        <span class='{res.style.nameSpan}' ui:field='nameSpan'/>
      </div>
    </div>

  </g:HTMLPanel>
</ui:UiBinder>
```

```
/**
 * Resources used by the entire application.
 */
public interface Resources extends ClientBundle {
  @Source("Style.css")
  Style style();

  @Source("Logo.jpg")
  ImageResource logo();

  public interface Style extends CssResource {
    String mainBlock();
    String nameSpan();
    Sprite userPictureSprite();
  }
}
```

```
// Within the owner class for the UiBinder template
@UiField Resources res;

...

res.style().ensureInjected();
```

The "with" element declares a field holding a resource object whose methods can be called to fill in attribute values. In this case it will be instantiated via a call to GWT.create(Resources.class). (Read on to see how pass an instance in instead of having it created for you.)

Note that there is no requirement that a ui:with resource implement the ClientBundle interface; this is just an example.

If you need more flexibility with a resource, you can set parameters on it with a <ui:attributes> element. Any setters or constructor arguments can be called on the resource object this way, just as for any other object in the template. In the example below, note how the FancyResources object accepts a reference to the Resource declared in the previous example.

```
public class FancyResources {
  enum Style {
    MOBILE, DESKTOP
  }

  private final Resources baseResources;
  private final Style style;

  @UiConstructor
  public FancyResources(Resources baseResources, Style style) {
    this.baseResources = baseResources;
    this.style = style;
  }
}
```

```
<ui:with field='fancyRes' type='com.my.app.widgets.logoname.FancyResources'>
  <ui:attributes style="MOBILE" baseResources="{res}"/>
</ui:with>
```

Share resource instances

You can make resources available to your template via the <ui:with> element, but at the cost of having them instantiated for you. If instead you want your code to be in charge of finding or creating that resource, you have two means to take control. You can mark a factory method with @UiFactory, or you can fill in a field yourself and annotate it as @UiField(provided = true).

Here's how to use @UiFactory to provide the Resources instance needed by the template in the previous example.

```
public class LogoNamePanel extends Composite {
  interface MyUiBinder extend UiBinder<Widget, LogoNamePanel> {}
  private static MyUiBinder uiBinder = GWT.create(MyUiBinder.class);

  @UiField SpanElement nameSpan;
  final Resources resources;

  public LogoNamePanel(Resources resources) {
    this.resources = resources;
    initWidget(uiBinder.createAndBindUi(this));
```

```
  }

  public void setUserName(String userName) {
    nameSpan.setInnerText(userName);
  }

  @UiFactory /* this method could be static if you like */
  public Resources getResources() {
    return resources;
  }
}
```

Any field in the template that is of type Resources will be instantiated by a call to `getResources`. If your factory method needs arguments, those will be required as attributes.

You can make things more concise, and have finer control, by using `@UiField(provided = true)`.

```
public class LogoNamePanel extends Composite {
  interface MyUiBinder extends UiBinder<Widget, LogoNamePanel> {}
  private static MyUiBinder uiBinder = GWT.create(MyUiBinder.class);

  @UiField SpanElement nameSpan;

  @UiField(provided = true)
  final Resources resources;

  public LogoNamePanel(Resources resources) {
    this.resources = resources;
    initWidget(uiBinder.createAndBindUi(this));
  }

  public void setUserName(String userName) {
    nameSpan.setInnerText(userName);
  }
}
```

Hello Text Resources

Now that we have resources, let's look back at the Hello world example at the top of the document. Having to write code just to get that name displayed was kind of cumbersome, especially if you're never going to change it. Instead, use `<ui:text>` to stitch it right into the template.

```
<ui:with field='res' type='com.my.app.widgets.logoname.Resources'/>

<div>
  Hello, <ui:text from='{res.userName}'/>.
</div>
```

Optimization Note: If that resource is coming from a method that the GWT compiler recognizes as doing nothing more than returning a compile time constant, any static final String, it will become part of the template itself through the magic of inlining — no additional function call will be made.

Hello Html Resources

Relying only on text as we did in the previous example is pretty limiting. Sometimes you have a fancy bit of markup to re-use, and it just doesn't need the full widget treatment. In such a case use `<ui:safehtml>` to stitch any SafeHtml into any HTML context.

```
<ui:with field='res' type='com.my.app.widgets.logoname.Resources'/>

<div>
  Hello, <ui:safehtml from='{res.fancyUserName}'/>.
</div>
```

You also have another option with HTML. Any SafeHtml class can be used directly, much the same as a widget.

```
<div>
  Hello, <my:FancyUserNameRenderer style="MOBILE">World</my:FancyUserNameRenderer>.
</div>
```

You might implement such a renderer this way. (Note the only slightly contrived use here of SafeHtmlTemplates to guard against XSS attacks.)

```
public class FancyUserNameRenderer implements SafeHtml, HasText {
  enum Style {
    MOBILE, DESKTOP
  }

  interface Templates extends SafeHtmlTemplates {
    @SafeHtmlTemplates.Template("<span class=\"mobile\">{0}</span>")
    SafeHtml mobile(String name);

    @SafeHtmlTemplates.Template("<div class=\"desktop\">{0}</div>")
    SafeHtml desktop(String name);
  }
  private static final Templates TEMPLATES = GWT.create(Templates.class);

  private final Style style;
  private String name;

  @UiConstructor
  public FancyResources(Style style) {
    this.style = style;
  }

  void setText(String text) {
    this.name = text;
  }

  @Override
  String asString() {
    switch (style) {
      Style.MOBILE: return TEMPLATES.mobile(name);
    }
    return Style.DESKTOP: return TEMPLATES.desktop(name);
  }
}
```

While this is a great technique, it's limited. Objects used this way are kind of a one-way-ticket. Their SafeHtml.asString() methods are called at render time (actually in most cases, at compile time thanks to inlining). Thus, if you access one through a `@UiField` in your java code, it won't have any handle to the DOM structure it created.

Apply different XML templates to the same widget

You're an MVP developer. You have a nice view interface, and a templated widget that implements it. How might you use several different XML templates for the same view?

Fair warning: This is only meant to be a demonstration of using different ui.xml files with the same code. It is not a proven pattern for implementing themes in an application, and may or may not be the best way to do that.

```
public class FooPickerController {
  public interface Display {
    HasText getTitleField();
    SourcesChangeEvents getPickerSelect();
  }

  public void setDisplay(FooPickerDisplay display) { ... }
}

public class FooPickerDisplay extends Composite
    implements FooPickerController.Display {

  @UiTemplate("RedFooPicker.ui.xml")
  interface RedBinder extends UiBinder<Widget, FooPickerDisplay> {}
  private static RedBinder redBinder = GWT.create(RedBinder.class);

  @UiTemplate("BlueFooPicker.ui.xml")
  interface BlueBinder extends UiBinder<Widget, FooPickerDisplay> {}
  private static BlueBinder blueBinder = GWT.create(BlueBinder.class);

  @UiField HasText titleField;
  @UiField SourcesChangeEvents pickerSelect;

  public HasText getTitleField() {
    return titleField;
  }
  public SourcesChangeEvents getPickerSelect() {
    return pickerSelect;
  }

  protected FooPickerDisplay(UiBinder<Widget, FooPickerDisplay> binder) {
    initWidget(uiBinder.createAndBindUi(this));
  }

  public static FooPickerDisplay createRedPicker() {
    return new FooPickerDisplay(redBinder);
  }

  public static FooPickerDisplay createBluePicker() {
    return new FooPickerDisplay(blueBinder);
  }
}
```

LazyPanel and LazyDomElement

You're trying to squeeze the last ounce of performance out of your application. Some of the widgets in your tab panel take a while to fire up, and they're not even visible to your user right away. You want to take advantage of LazyPanel. But you're feeling lazy yourself: it's abstract, and you really

don't want to deal with extending.

```
<gwt:TabLayoutPanel barUnit='EM' barHeight='1.5'>
  <gwt:tab>
    <gwt:header>Summary</gwt:header>
    <gwt:LazyPanel>
      <my:SummaryPanel/>
    </gwt:LazyPanel>
  </gwt:tab>
  <gwt:tab>
    <gwt:header>Profile</gwt:header>
    <gwt:LazyPanel>
      <my:ProfilePanel/>
    </gwt:LazyPanel>
  </gwt:tab>
  <gwt:tab>
    <gwt:header>Reports</gwt:header>
    <gwt:LazyPanel>
      <my:ReportsPanel/>
    </gwt:LazyPanel>
  </gwt:tab>
</gwt:TabLayoutPanel>
```

That helped, but there is more you can do.

Elsewhere in the app is a template with a lot of dom element fields. You know that when your ui is built, a `getElementById()` call is made for each and every one of them. In a large page, that can add up. By using LazyDomElement you can defer those calls until they're actually needed — if they ever are.

```
public class HelloWorld extends UIObject { // Could extend Widget instead
  interface MyUiBinder extends UiBinder<DivElement, HelloWorld> {}
  private static MyUiBinder uiBinder = GWT.create(MyUiBinder.class);

  @UiField LazyDomElement<SpanElement> nameSpan;

  public HelloWorld() {
    // createAndBindUi initializes this.nameSpan
    setElement(uiBinder.createAndBindUi(this));
  }

  public void setName(String name) { nameSpan.get().setInnerText(name); }
}
```

Rendering HTML for Cells

Cell Widgets require the generation of HTML strings, but writing the code to concatenate strings to form proper HTML quickly gets old. UiBinder lets you use the same templates to render that HTML.

```
<!-- HelloWorldCell.ui.xml -->

<ui:UiBinder xmlns:ui='urn:ui:com.google.gwt.uibinder'>
  <ui:with field='name' type='java.lang.String'/>

  <div>
    Hello, <span><ui:text from='{name}'/></span>.
```

```
  </div>
</ui:UiBinder>
```

The `<ui:with>` tag defines fields to use as data to render the template. These templates can only contain HTML elements, no widgets or panels.

Now, define a `HelloWorldCell` widget. Add an interface that extends the UiRenderer interface (instead of `UiBinder`).

```
public class HelloWorldCell extends AbstractCell {
  interface MyUiRenderer extends UiRenderer {
    void render(SafeHtmlBuilder sb, String name);
  }
  private static MyUiRenderer renderer = GWT.create(MyUiRenderer.class);

  @Override
  public void render(Context context, String value, SafeHtmlBuilder builder) {
    renderer.render(builder, value);
  }
}
```

UiBinder uses the names of the parameters in `MyUiRenderer.render()` to match the fields defined in `<ui:with>` tags. Use as many as needed to render your data.

Cell event handling with UiBinder

Cell events require you to write the code to determine the exact cell on which the event was received, and even more. UiBinder handles a lot of this work for you. It will route events to your handler methods based on the event type, and the HTML element on which the event was received.

Taking the HelloWorldCell.ui.xml template, let's handle click events in the "name" `<span>` element.

First, add a `ui:field` attribute to the `<span>`. This allows the generated code to distinguish the span element from the other elements in the template.

```
<div>
  Hello, <span ui:field='nameSpan'><ui:text from='{name}'/></span>.
</div>
```

Add a `onBrowserEvent` method to `MyUiRenderer`. The `onBrowserEvent` in the renderer interface only requires the first three arguments to be defined. Any other arguments after that are for your convenience, and will be passed verbatim to the handlers. The first argument is what UiRenderer uses to dispatch events from `onBrowserEvent` to methods in a Cell Widget object.

```
interface MyUiRenderer extends UiRenderer {
  void render(SafeHtmlBuilder sb, String name);
  onBrowserEvent(HelloWorldCell o, NativeEvent e, Element p, String n);
}
```

Let the AbstractCell know that you will handle `click` events.

```
public HelloWorldCell() {
  super("click");
}
```

Let the Cell `onBrowserEvent` delegate the handling to the `renderer`.

```
@Override
public void onBrowserEvent(Context context, Element parent, String value,
    NativeEvent event, ValueUpdater<String> updater) {
  renderer.onBrowserEvent(this, event, parent, value);
}
```

Finally, add the handler method to `HelloWorldCell`, and tag it with `@UiHandler({"nameSpan"})`. The type of the first parameter, ClickEvent, will determine the type of event handled.

```
@UiHandler({"nameSpan"})
void onNameGotPressed(ClickEvent event, Element parent, String name) {
  Window.alert(name + " was pressed!");
}
```

Getting rendered elements

Once a cell is rendered it is possible to retrieve and operate on elements marked with `ui:field`. This is useful when you need to manipulate the DOM elements.

```
interface MyUiRenderer extends UiRenderer {
  // ... snip ...
  SpanElement getNameSpan(Element parent);
  // ... snip ...
}
```

Use the getter by passing the parent element received by the Cell widget. The name of these getters must match the `ui:field` tags placed on the template, prepended with "get". That is, for a `ui:field` named `someName`, the getter should be `getSomeName(Element parent)`.

```
@UiHandler({"nameSpan"})
void onNameGotPressed(ClickEvent event, Element parent, String name) {
  renderer.getNameSpan(parent).setInnerText(name + ", dude!");
}
```

Access to styles with UiRenderers

UiRenderer lets you define getters to retrieve styles defined in the template. Just define a getter with no parameters matching the style name and returning the style type.

```
<ui:style field="myStyle" type="com.my.app.AStyle">
  .red {color:#900;}
  .normal {color:#000;}
</ui:style>

<div>
  Hello, <span ui:field="nameSpan" class="{myStyle.normal}">
    <ui:text from="{name}"/></span>.
</div>
```

Define the style interface:

```
public interface AStyle extends CssResource {
  String normal();
  String red();
}
```

Define the style getter in the UiRenderer interface, prepending the style name with "get":

```
interface MyUiRenderer extends UiRenderer {
  // ... snip ...
  AStyle getMyStyle();
  // ... snip ...
}
```

Then use the style wherever you need it. Notice that you need to get the style name using the `red()` accessor method. The GWT compiler obfuscates the actual name of the style to prevent collisions with other similarly named styles in your application.

```
@UiHandler({"nameSpan"})
void onNameGotPressed(ClickEvent event, Element parent, String name) {
  String redStyle = renderer.getMyStyle().red();
  renderer.getNameSpan(parent).replaceClass(redStyle);
}
```